程式語言不等於文字：結構、語意與執行的基本分離

A Programming Language Is Not Text: The Fundamental Separation of Structure, Semantics, and Execution

v0.1 / 2026-07-27 / Neo.K with Aletheia / 初版完成

https://thisoneisneok.com/html/papers/pu-2-01.html

SECTION

A Programming Language Is Not Text: The Fundamental Separation of Structure, Semantics, and Execution

論文編號： PU-2-01

系列：「程式宇宙書系」第 2 冊《程式語言的本質》

作者： Neo.K with Aletheia

機構： EveMissLab／一言諾科技有限公司

版本： v0.1

日期： 2026 年 7 月 27 日

SECTION

摘要

前兩冊已分別回答：為何需要計算、計算世界與物理世界如何耦合，以及程式應如何被建模、切分、理解、驗證與維護。本冊轉向更深一層問題：程式語言究竟是什麼？

現代程式語言通常以文字形式出現。開發者在編輯器中輸入字元，編譯器或解譯器解析字串，形成抽象語法樹，再轉換為中介表示、機器碼或 Runtime 行為。由於文字是最可見、最常操作、最容易儲存與版本控制的表面，程式語言長期被誤認為一種特殊文字。

本文提出：

> 【Programming Language
> ≠
> Text】

文字只是程式語言的一種序列化投影。程式語言更根本的存在結構是：

> 【mathcal L_P
> =
> <
> Σ,
> Γ,
> mathcal S,
> mathcal M,
> mathcal E,
> mathcal V,
> mathcal R
> >】

其中：

- Σ：符號與基本構件；
- Γ：組合與形成規則；
- mathcal S：結構空間；
- mathcal M：語意映射；
- mathcal E：效果與世界差分；
- mathcal V：合法性、驗證與不變量；
- mathcal R：Runtime／執行解釋。

本文將程式語言分為三個不可互相化約的核心層：

> 【Language
> =
> Structure
> +
> Semantics
> +
> Execution】

其中：

- 結構 Structure：哪些構件能以何種關係組成合法程式；
- 語意 Semantics：這些結構在問題世界中代表什麼；
- 執行 Execution：這些語意如何被 Runtime 轉化為狀態、事件與外部效果。

  文字只主要承擔結構的線性序列化。即使兩段文字不同，只要它們形成同一結構、保持同一語意並產生相容效果，就可能是同一程式的不同表達。反之，外觀相同的文字若置於不同作用域、版本、型別環境、權限、Runtime 或外部世界中，也可能具有不同語意與效果。

  本文因此提出「三重同一性」：

> SameProgram
> =
> StructuralEquivalence
> ∧
> SemanticEquivalence
> ∧
> EffectEquivalence

文本相同只提供：

> TextIdentity

而：

> 【TextIdentity
> not⇒
> ProgramIdentity】

本文進一步區分：

- 字元；
- token；
- 語法節點；
- 抽象語法樹；
- 語意節點；
- 中介表示；
- 效果圖；
- 執行計畫；
- Runtime 行為；
- 世界差分。

這些並不是同一對象的不同名稱，而是程式在不同投影與處理階段中的不同存在。

本文提出「文本只是傳輸層」命題。文字適合：

- 線性輸入；
- 精確編輯；
- 差異比較；
- 儲存；
- 傳輸；
- 搜尋；

- 版本控制。

  但文字不天然適合直接呈現：

- 多重關係；
- 並行；
- 狀態機；
- 權限；
- 依賴；
- 時間；
- 多角色投影；
- 失敗與補償；
- 世界邊界。

  因此，大量註解、命名規則、資料夾、設計模式與文件，往往是在替文字投影補回其無法直接顯示的結構。

  本文亦提出「結構先於文字」原則：

> 【Intent
> →
> Semantic Structure
> →
> Executable Structure
> →
> Textual Projection】

這不是要求所有程式語言取消文字，而是重新排列其本體順序：文字應是可編譯結構的一種編輯與交換介面，而不是程式存在的唯一權威來源。

本文分析文字語言、視覺語言、AST 編輯器、表單式 DSL、工作流、狀態圖、資料流語言與 AI 意圖介面，指出真正差異不在於是否使用文字，而在於：

- 哪一層是權威結構；

- 哪些語意被直接表示；
- 哪些內容必須靠推論；
- 哪些效果能被驗證；
- 哪些結構可被多重投影；
- 哪些修改能保持語意身份。

本文也處理自然語言與程式語言的界線。自然語言具有高度意圖表達力，卻常存在歧義、上下文依賴與不可直接執行；傳統程式語言具有精確語法與執行性，卻要求人類先把問題壓縮為機器可接受結構。未來程式語言不必在兩者間二選一，而可建立：

> 【Intent Layer
> →
> Verified Semantic IR
> →
> Multiple Executable Projections】

本文使用條件判斷、資料查詢、工作流、UI、AI Agent 與物理裝置控制等案例，說明同一意圖如何具有多種文字與非文字投影。本文最後提出可證偽研究綱領，包括文本—結構漂移、跨表示語意保持、AST 編輯效率、語意縮放、程式同一性判定、自然語言到 IR 的歧義率、視覺語言規模限制、結構化版本控制與 AI 生成程式可驗證性。

本文為下一篇〈符號作為算子〉奠定起點：既然程式語言不是文字，那麼符號就不應只被理解為靜態字元，而應被理解為帶有輸入、輸出、前置條件、效果、組合律與驗證規則的算子。

關鍵詞： 程式語言本體論、文字投影、結構先於文字、語意、中介表示、AST、後文本程式語言、意圖語言

SECTION

Abstract

This paper argues that a programming language is not text. Text is only one serialization and editing projection of a deeper language structure.

A programming language is modeled as:

> mathcal L_P
> =
> <
> Σ,
> Γ,
> mathcal S,
> mathcal M,
> mathcal E,
> mathcal V,
> mathcal R
> >

The paper separates structure, semantics, and execution; distinguishes text identity from program identity; and analyzes tokens, syntax trees, semantic nodes, intermediate representations, effect graphs, execution plans, and world differences as distinct layers.

It proposes that intent should be translated into verified semantic structure and then projected into one or more textual, visual, or executable representations.

Keywords: programming language ontology, text projection, structure before text, semantics, intermediate representation, post-textual programming

開發者通常透過文字接觸程式語言。

例如：

```
if balance >= amount:
  balance -= amount
```

肉眼看到的是：

- 字元；
- 空格；
- 換行；
- 關鍵字；
- 標點；

- 變數名稱。

  但真正執行時，Runtime 並不是直接理解「這一串字」。

  它通常需要經過：

  文字
  → token
  → 語法結構
  → 名稱解析
  → 型別與效果檢查
  → 中介表示
  → 執行計畫
  → Runtime 行為
  所以文字是入口，不是全部本體。

  本文定義：

> 【mathcal L_P
> =
> <
> Σ,
> Γ,
> mathcal S,
> mathcal M,
> mathcal E,
> mathcal V,
> mathcal R
> >】

SECTION

2.1 符號集合 Σ

包含：

- 關鍵字；
- 名稱；

- 常值；
- 操作符；
- 型別；
- 節點；
- 圖形構件；
- 結構標籤。

SECTION

2.2 形成規則 Γ

決定哪些構件能以何種方式組合。

SECTION

2.3 結構空間 $\mathcal{S}$

合法程式結構的集合。

SECTION

2.4 語意映射 $\mathcal{M}$

把結構映射到：

- 值；
- 狀態；
- 關係；
- 行動；
- 事件；

- 世界意義。

SECTION

2.5 效果集合 mathcal E

描述程式可能：

- 讀寫狀態；
- 呼叫外部工具；
- 產生事件；
- 修改權限；
- 改變物理世界。

SECTION

2.6 驗證系統 mathcal V

包含：

- 型別；
- 不變量；
- 前置條件；
- 權限；
- 效果限制；
- 組合合法性。

SECTION

2.7 Runtime mathcal R

決定結構如何被實際解釋與執行。

本文提出：

> 【Language
> =
> Structure
> +
> Semantics
> +
> Execution】

SECTION

3.1 結構

回答：

哪些東西可以組合？
如何形成合法表達？

SECTION

3.2 語意

回答：

這個結構代表什麼？
它引用哪個世界對象？

SECTION

3.3 執行

回答：

Runtime 如何把此結構轉成狀態變化與效果？

SECTION

3.4 三者分離的必要

語法合法的程式可能沒有合法語意。

語意合法的程式可能未被授權執行。

可執行的程式也可能產生不符合原始目的的效果。

因此：

> StructuralValidity
> not⇒
> SemanticValidity
> not⇒
> ExecutionLegitimacy

SECTION

4.1 線性化

問題世界與程式結構通常具有：

- 樹；
- 圖；
- 網路；
- 多重依賴；
- 並行；
- 時間；
- 作用域。

  文字則主要是線性序列：

> T
> =
> [
> c⊠,c⊠,…,c⊠
>]

因此，文字表示必須把非線性結構序列化。

SECTION

4.2 序列化規則

縮排、括號、分隔符、名稱與關鍵字，都是用線性符號重建結構的方法。

SECTION

4.3 文本投影

設程式結構為 P_s，則文字投影為：

> T_L
> =
> Serialize_L
> (
> P_s
>)

解析則為：

> Parse_L
> (
> T_L
>)
> =
> P_s'

理想上：

> P_s'
> cong
> P_s

SECTION

4.4 投影損失

文字通常不能直接保存所有：

- 設計理由；
- 問題世界身份；
- 權限；
- 外部證據；
- 圖形位置；
- 多角色視圖；
- Runtime 觀測；
- 歷史來源。

這些內容常被放入註解、文件或外部工具。

SECTION

4.5 格式不是語意

不同空格、換行與括號風格，可能形成相同結構。

因此：

> TextDifference
> not⇒
> ProgramDifference

程式的典型轉換鏈可表示為：

> Characters
> →
> Tokens

SECTION

5.1 字元 Characters

純粹編碼單位。

SECTION

5.2 Token

具有詞法類型的符號單位，例如：

- identifier；
- number；
- operator；
- keyword。

SECTION

5.3 語法節點 Syntax Node

表達結構關係，例如：

- 條件；
- 呼叫；
- 宣告；
- 區塊；
- 型別。

SECTION

5.4 語意結構 Semantic Structure

將名稱與問題世界對象、作用域、型別、權限與效果連接。

SECTION

5.5 中介表示 IR

為分析、驗證、最佳化或多後端生成建立的結構。

SECTION

5.6 執行計畫

確定：

- 順序；
- 資源；
- 並行；
- 工具；
- 交易；
- 失敗處理。

SECTION

5.7 效果

Runtime 實際讀寫狀態、發布事件或調用外部世界。

SECTION

5.8 世界差分

$$\Delta W = W_{t+1}$$

最終程式價值不在於文字被解析，而在於世界是否以合法方式改變。

SECTION

6.1 Token 不等於語法節點

同一 token 可以在不同結構中具有不同位置。

SECTION

6.2 語法節點不等於語意節點

名稱 user 只有在名稱解析與領域連結後，才知道指向哪個對象。

SECTION

6.3 AST 不等於 IR

AST 通常保留來源語言結構；IR 可重新組織、正規化或降低抽象層次。

SECTION

6.4 IR 不等於執行

IR 仍可能：

- 未排程；
- 未配置資源；
- 未授權；
- 未部署；
- 未連接外部服務。

SECTION

6.5 執行不等於目的完成

Runtime 成功執行，不代表世界效果或原始意圖完成。

本文提出：

> SameProgram
> =
> StructuralEquivalence
> ∧
> SemanticEquivalence
> ∧
> EffectEquivalence

SECTION

7.1 結構等價

兩個表示形成相容的抽象結構。

SECTION

7.2 語意等價

它們指向相同問題世界概念、規則與狀態。

SECTION

7.3 效果等價

在明示條件下，它們產生相容世界差分與失敗語意。

SECTION

7.4 文本同一性不足

同一份文字置於不同：

- 作用域；

- 套件版本；
- 編譯器；
- 設定；
- 資料；
- 權限；
- Runtime；

  可能產生不同效果。

  所以：

> 【TextIdentity
> not⇒
> ProgramIdentity】

SECTION

7.5 文本差異也不足

重命名變數、重新格式化或改寫控制流，不必改變程式語意。

SECTION

8.1 Source of Truth

傳統專案常把文字來源碼視為唯一權威。

SECTION

8.2 其他可能權威

程式也可能以以下結構為權威：

- schema；

- 狀態機；
- 規則表；
- AST；
- Intent IR；
- 工作流圖；
- 形式規格；
- 可視節點圖。

SECTION

8.3 生成式來源碼

若來源碼由更高層結構生成，手動修改生成檔會造成投影分裂。

SECTION

8.4 權威層必須明示

完整工具鏈應說明：

哪一層可以被人類修改？
哪一層可被重新生成？
哪一層保存語意身份？
衝突時以何者為準？

SECTION

8.5 多權威風險

若文字、圖形、schema 與自然語言規格都能獨立修改，卻沒有同步協議，系統會產生多個互相競爭的真相。

本文並不否定文字。

SECTION

9.1 高密度

熟練使用者可以快速輸入複雜結構。

SECTION

9.2 精確

字元序列易於解析、比較與重現。

SECTION

9.3 可版本控制

文字差異工具成熟。

SECTION

9.4 易搜尋

名稱與模式可被快速查詢。

SECTION

9.5 易傳輸

幾乎所有系統都能處理文字。

SECTION

9.6 可組合工具鏈

編輯器、編譯器、格式化、靜態分析與建置系統高度成熟。

因此，問題不是取消文字，而是停止把文字誤認為全部程式結構。

SECTION

10.1 關係隱藏

跨檔案、跨模組與跨時間關係通常需由工具重新推導。

SECTION

10.2 多尺度困難

文字瀏覽器不容易同時呈現世界、模組、契約與 Runtime。

SECTION

10.3 並行序列化

並行結構常被迫寫成某種線性順序。

SECTION

10.4 權限與效果隱藏

許多語言不能直接從語法看出外部效果、授權與不可逆性。

SECTION

10.5 失敗散落

失敗可能分散於：

- exception；
- callback；
- retry；

- middleware；
- queue；
- 人工流程。

SECTION

10.6 設計理由缺席

來源碼通常表達「如何」，不完整表達「為何」。

SECTION

11.1 視覺化的優勢

視覺語言可以直接表達：

- 節點；
- 邊；
- 狀態；
- 流程；
- 依賴；
- 並行；
- 分支。

SECTION

11.2 視覺化的限制

當節點數量上升，可能產生：

- 畫布爆炸；

- 連線交叉；
- 精確編輯困難；
- 版本差異不易閱讀；
- 鍵盤效率下降；
- 抽象能力不足。

SECTION

11.3 視覺不等於結構

圖形介面若只是把文字函式變成方塊，未必真正改變語言本體。

SECTION

11.4 混合語言

成熟語言可以讓：

- 文字編輯局部結構；
- 圖形呈現全域關係；
- 表格編輯規則；
- 狀態圖編輯流程；
- 自然語言描述意圖；
- IR 保存權威語意。

SECTION

11.5 投影同步

不同投影必須由共同結構生成：

> V_(text)
> =
> π_(text)
> (
> S^ast
>)

> V_(visual)
> =
> π_(visual)
> (
> S^ast
>)

其中 S^ast 為共同權威結構。

SECTION

12.1 文字編輯

文字編輯允許暫時產生不合法中間狀態。

例如輸入括號過程中，程式短暫無法解析。

SECTION

12.2 結構化編輯

結構化編輯直接操作：

- 節點；
- 關係；
- 型別；
- 作用域；

- 效果；
- 契約。

SECTION

12.3 優點

- 非法結構可被限制；
- 重構更安全；
- 節點身份可跨修改保存；
- 多投影同步較容易；
- AI 可直接修改語意節點。

SECTION

12.4 代價

- 自由輸入受限；
- 不完整想法難以暫存；
- 工具複雜；
- 大量細節操作可能笨重；
- 使用者需理解結構。

SECTION

12.5 半結構狀態

理想編輯器應允許：

hole
unknown

draft-node
unresolved-reference
type-to-be-inferred
使不完整意圖能存在，而不必退回無結構文字。

SECTION

13.1 未完成不等於錯誤

程式設計本來就是逐步形成結構的過程。

SECTION

13.2 Hole

定義：

> Box_T

表示需要某型別或語意的未完成位置。

SECTION

13.3 Hole 的資訊

可保存：

- 預期型別；
- 可用變數；
- 前置狀態；
- 允許效果；
- 未決問題；
- 來源意圖。

SECTION

13.4 AI 補全

AI 不應只猜下一段文字，而可針對 Hole 生成候選結構，並附：

- 語意；
- 依賴；
- 效果；
- 風險；
- 驗證條件。

SECTION

13.5 未完成程式作為合法狀態

這使程式工具能正式處理「正在思考」的結構，而非只接受完整可編譯文字。

SECTION

14.1 自然語言的優勢

自然語言擅長表達：

- 目的；
- 背景；
- 例外；
- 價值；
- 模糊需求；
- 尚未決定內容。

SECTION

14.2 自然語言的限制

它具有：

- 歧義；
- 省略；
- 上下文依賴；
- 代詞；
- 隱含常識；
- 不一致；
- 難以直接驗證。

SECTION

14.3 程式語言的優勢

傳統程式語言要求明確：

- 名稱；
- 結構；
- 型別；
- 順序；
- 作用域；
- 操作。

SECTION

14.4 程式語言的限制

人類必須先完成大量轉譯，把世界與意圖壓縮為機器結構。

SECTION

14.5 不是互相取代

未來架構可以是：

> 【NaturalIntent
> →
> SemanticIR
> →
> Verification
> →
> ExecutableProjection】

自然語言提供意圖與背景；結構化 IR 消除歧義、保留責任並建立可驗證執行。

SECTION

15.1 原始意圖

例如：

當付款真正成功後，
把訂單標成已付款，
但不要因重複通知而重複處理。

SECTION

15.2 語意展開

需要解析：

- 付款真正成功的證據；
- 訂單權威狀態；
- 事件身份；

- 冪等；
- 前置條件；
- 失敗與未知結果；
- 責任模組。

SECTION

15.3 中介結構

```
intent_ir:
  action: "record-payment"
  trigger: "PaymentAuthorized"
  target: "Order"
  precondition: "order.status == payment-pending"
  idempotency: "payment-event-id"
  effect: "order.status = paid"
  unknown_outcome: "human-reconciliation"
```

SECTION

15.4 生成投影

同一 IR 可以生成：

- 程式碼；
- 狀態圖；
- 測試；
- 契約；
- 文件；
- 監控；
- 稽核規則。

SECTION

15.5 人類確認

涉及價值、權限與不可逆效果時，編譯器不應只追求自動完成，而應要求明示確認。

SECTION

16.1 為何需要 IR

IR 可以作為自然意圖與具體語言之間的穩定橋梁。

SECTION

16.2 語意 IR 應保存

- 問題世界身份；
- 實體與關係；
- 狀態；
- 行動；
- 事件；
- 規則；
- 權限；
- 效果；
- 失敗；
- 證據；
- 來源意圖。

SECTION

16.3 IR 不應只是低階指令

傳統 IR 主要服務編譯最佳化。

意圖時代的 IR 還應服務：

- 語意保持；
- 多後端；
- 人類理解；
- AI 修改；
- 驗證；
- 治理；
- 世界差分分析。

SECTION

16.4 IR 版本

語意 IR 也需要：

- schema；
- 語意版本；
- 遷移；
- 相容；
- 歷史；
- 差異。

SECTION

17.1 傳統隱藏效果

函式名稱與回傳型別常無法看出：

- 寫資料庫；
- 寄信；
- 付款；
- 發布；
- 修改權限；
- 控制設備。

SECTION

17.2 效果標記

語言可明示：

```
reads Order
writes Payment
emits PaymentAuthorized
calls ExternalBank
requires HumanApproval
irreversible SendMessage
```

SECTION

17.3 效果驗證

編譯器或 Runtime 可檢查：

- 是否有權；
- 是否超出模組責任；
- 是否需要交易；

- 是否需冪等；
- 是否可回復；
- 是否需外部證據。

SECTION

17.4 世界差分預覽

執行前可產生：

> widehatΔ W

顯示預期影響與風險。

SECTION

18.1 傳統作用域

決定變數名稱在哪裡可引用。

SECTION

18.2 世界作用域

未來語言還可描述：

- 哪個問題世界；
- 哪個模組責任；
- 哪個身份；
- 哪個權限；
- 哪個時間；
- 哪個環境；

- 哪個資料敏感域。

SECTION

18.3 效果作用域

某一操作只能作用於明示資源與範圍。

SECTION

18.4 Agent 作用域

Agent 生成的程式必須繼承：

- 工具權限；
- 資料作用域；
- 預算；
- 時間；
- 不可逆限制。

SECTION

19.1 多投影問題

同一程式可能同時具有：

- 文字；
- 圖形；
- 表格；
- 狀態圖；
- 自然語言說明；

- Runtime 設定。

SECTION

19.2 修改傳播

若使用者在視覺圖中修改節點，其他投影應由共同結構重新生成。

> S^ast
> xrightarrowπ□
> V□

> S^ast
> xrightarrowπ□
> V□

而不是：

> V□
> ↔
> V□

透過大量脆弱雙向轉換互相同步。

SECTION

19.3 Round-Trip

理想上：

> Parse
> (
> Print
> (
> S^ast
>)
>)

但註解、格式、位置與非語意資訊可能需要額外保存。

SECTION

19.4 修改衝突

若兩個投影同時修改，應進行：

- 節點身份比較；
- 語意差分；
- 衝突定位；
- 人類選擇；
- 分支與合併。

SECTION

19.5 語意差分

傳統文字 diff 顯示字元變化。

語意 diff 應顯示：

- 新增何種世界狀態；
- 改變哪條規則；
- 新增何種效果；
- 哪個契約被破壞；
- 哪些主體受到影響。

SECTION

20.1 傳統理解

編譯通常被理解為：

> Source
> →
> Target

SECTION

20.2 擴展理解

意圖時代的編譯應包含：

> 【Compile
> =
> Interpret
> +
> Disambiguate
> +
> Validate
> +
> Authorize
> +
> Lower
> +
> Generate
> +
> Verify】

SECTION

20.3 Interpret

理解結構所指向的問題世界意義。

SECTION

20.4 Disambiguate

把未決、歧義與衝突轉化為選擇點。

SECTION

20.5 Validate

檢查型別、狀態、規則、權限與效果。

SECTION

20.6 Authorize

確認生成與執行是否在授權範圍。

SECTION

20.7 Lower

將高層語意逐步降低為可執行結構。

SECTION

20.8 Generate

生成程式碼、配置、工作流、測試與可視投影。

SECTION

20.9 Verify

驗證投影保持原始語意與邊界。

原始文字：

```
if age >= 18:
  allow()
```

文字沒有直接回答：

- age 來自哪裡；
- 何時測量；
- 18 是法律、政策還是暫時規則；
- allow 允許什麼；
- 是否有例外；
- 誰可修改門檻。

語意結構應進一步表示：

```
rule:
 subject: "applicant"
 property: "verified-age"
 threshold: 18
 authority: "policy-v3"
 effect: "grant-standard-access"
 exceptions:
  - "guardian-approved"
```

因此，文字條件只是規範結構的低階投影。

SQL：

```
SELECT * FROM orders WHERE status = 'paid';
```

文字結構清楚，但仍需知道：

- orders 是權威狀態還是投影；
- paid 的語意；
- 是否包含結算未完成；
- 查詢者是否有權看全部訂單；
- 時間與新鮮度；

- 是否包含已刪除或爭議資料。

  同一 SQL 在不同 schema 與治理下，代表不同程式。

  文字可以寫出步驟：

  approve
  deploy
  verify
  publish
  但工作流圖更容易直接表示：

- 分支；
- 並行；
- 等待；
- 超時；
- 補償；
- 人工節點；
- 回復。

  理想語言可讓同一工作流結構投影成：

- 視覺圖；
- YAML；
- 程式碼；
- 執行引擎設定；
- 事故回放。

  UI 也可以被視為程式語言投影。

  按鈕、表單與流程表示：

- 可行動集合；

- 輸入型別；
- 狀態；
- 權限；
- 預設；
- 操作效果。

若 UI、後端契約與世界狀態來自不同結構，便容易產生：

- 畫面顯示可做但後端拒絕；
- 後端支援但 UI 無入口；
- 成功文案與真實完成層級不同。

共同語意 IR 可以生成 UI 與後端契約，降低此類漂移。

使用者說：

> 把網站更新後發布。

Agent 不能直接把自然語言當成已完成程式。

它需要展開：

- 哪個網站；
- 更新內容；
- 測試；
- 發布環境；
- 權限；
- 是否需要批准；
- 回滾；
- 成功證據。

Agent 應產生可審核 Intent IR，而不是直接生成並執行文字指令。

文字：

```
motor.start()
```

無法單獨表達：

- 哪台馬達；
- 目前物理狀態；
- 安全條件；
- 人員是否在危險區；
- 最大轉速；
- 停止路徑；
- 感測器是否可信；
- 命令送出是否等於馬達啟動。

  物理耦合語言必須把效果、感測、授權與安全不變量提升為一等結構。

- 文字本體化： 把字元序列當成程式語言全部。
- 語法即語意： 可解析便宣稱具有正確世界意義。
- 語意即執行： 有明確意義便假定可合法執行。
- 執行即目的： Runtime 完成便宣稱原始意圖完成。
- 文本同一即程式同一： 忽略作用域、版本、權限與 Runtime。
- 文本差異即程式差異： 把格式與重命名當成語意變更。
- AST 即完整本體： 忽略領域身份、效果、權限與證據。
- IR 只服務最佳化： 缺少意圖、責任與世界語意。
- 單一來源真相迷思： 多投影能獨立修改卻無同步協議。
- 生成檔手改： 修改低階投影造成權威結構分裂。

- 視覺語言萬能論： 畫布節點增加後失去可讀性與精確性。
- 文字語言萬能論： 把所有非線性結構交給人腦推導。
- 自然語言直接執行： 未消除歧義、權限與不可逆風險。
- AI 補全文字化： 只預測下一 token，不理解結構 Hole。
- 不允許未完成結構： 使用者只能在完整程式與純文字之間切換。
- 效果隱藏： 函式看似純粹，實際修改外部世界。
- 權限外掛化： 授權完全位於語言結構之外。
- 作用域狹化： 只管理名稱，不管理世界、資料與效果範圍。
- 文字 diff 僭位： 字元變更被當成完整影響分析。
- 投影漂移： 文字、圖、契約、UI 與 Runtime 使用不同語意。
- 編譯只是轉碼： 忽略消歧、驗證、授權與效果證明。
- 物理命令等於物理效果： 發出控制指令便宣稱裝置已完成操作。

SECTION

28.1 文本—結構漂移率

比較文字表示與權威 AST／IR：

> R_(TS)
> =
> (|semantic structures not faithfully represented in text|)/(|sampled semantic structures|)

SECTION

28.2 跨表示語意保持

將同一權威結構投影為文字、圖形與表格，再由不同投影重建，測量：

- 結構；
- 語意；
- 效果；
- 失敗；

  的保持程度。

SECTION

28.3 程式同一性判定

比較：

- 文字 diff；
- AST diff；
- 語意 diff；
- 效果 diff；

  對真實變更影響的預測準確率。

SECTION

28.4 結構化編輯效率

測量文字編輯與結構化編輯在：

- 輸入速度；
- 非法狀態；
- 重構；
- 大型變更；
- 學習成本；

上的差異。

SECTION

28.5 Hole 輔助效果

比較一般文字補全與帶有型別、作用域、效果及意圖的結構 Hole，在 AI 生成正確率上的差異。

SECTION

28.6 視覺語言規模限制

測量節點數、邊數與嵌套深度增加後，視覺編輯的理解速度與錯誤率。

SECTION

28.7 混合投影效益

比較純文字、純視覺與共同 IR 多投影工具，在不同任務與角色上的表現。

SECTION

28.8 自然語言歧義率

統計自然語言意圖轉換成語意 IR 時，需要：

- 補充資訊；
- 人類選擇；
- 權限確認；
- 例外定義；

  的比例。

SECTION

28.9 IR 語意完整度

檢查 IR 是否保存：

- 世界身份；
- 狀態；
- 行動；
- 規則；
- 權限；
- 效果；
- 失敗；
- 證據；
- 原始意圖。

SECTION

28.10 投影同步錯誤率

統計文字、圖形、UI、契約與 Runtime 配置之間因獨立修改造成的漂移。

SECTION

28.11 效果標記與事故率

研究明示外部效果、不可逆性與授權需求的語言，是否降低越權與重複操作事故。

SECTION

28.12 語意差分價值

比較字元 diff 與語意 diff 對程式審查、影響分析與事故預防的效果。

SECTION

28.13 AI 生成可驗證性

比較 AI 直接生成來源碼，以及先生成可驗證語意 IR 再投影來源碼的正確性、可解釋性與恢復性。

SECTION

28.14 物理耦合語言

測量明示感測、命令、效果確認與安全不變量的語言，在裝置控制中的失敗辨識能力。

- 程式語言不等於文字。
- 文字是程式結構的一種線性序列化投影。
- 程式語言至少包含符號、形成規則、結構、語意、效果、驗證與 Runtime。
- 結構合法不代表語意合法，語意合法不代表執行正當。
- 字元、token、AST、語意節點、IR、執行計畫與世界差分是不同層級。
- 文本同一不保證程式同一。
- 文本差異不必然造成程式語意差異。
- 程式同一性需要結構、語意與效果等價。
- 文字具有高密度、精確、可搜尋與版本控制優勢，但不天然呈現多重關係與效果。
- 視覺語言能直接表達關係，卻不自動優於文字語言。
- 多種編輯投影應由共同權威結構生成。
- 結構化編輯應允許 Hole、未知與未完成語意。
- AI 程式補全應針對語意結構，而非只預測下一段文字。
- 自然語言適合表達意圖與背景，但不能未經消歧與驗證直接取得執行權。
- 語意 IR 應成為意圖與多種可執行投影之間的穩定橋梁。

- 未來 IR 不只服務最佳化，也服務語意保持、治理、驗證與多後端生成。
- 外部效果、權限、不可逆性與證據應成為語言的一等結構。
- 作用域應從名稱可見性擴展至世界、資料、效果、時間與權限範圍。
- 語意差分比字元差分更接近真實程式變更。
- 編譯應包含理解、消歧、驗證、授權、降低、生成與證明。
- 來源碼可以繼續存在，但不應永久壟斷程式的權威本體。

22.

> 【程式語言真正表示的，】

> 【不是一串等待機器閱讀的文字，】

> 【而是一個能被多重投影、驗證並執行的】

> 【世界結構。】

SECTION

30.1 承接第 0 冊

第 0 冊指出意圖、計算與物理世界互不等同，任何表示都具有落差。

本篇進一步指出，文字只是意圖與計算結構之間的其中一層表示，不能冒充完整程式本體。

SECTION

30.2 承接第 1 冊

第 1 冊已建立：

- 問題世界；

- 狀態動力；
- 責任模組；
- 契約；
- 外部結構記憶；
- 失敗與恢復。

本篇指出，未來程式語言應直接表示這些結構，而不是把它們全部壓縮進文字慣例、框架與人腦。

SECTION

30.3 銜接 PU-2-02

下一篇〈符號作為算子〉將研究：

- 符號如何從字元提升為操作結構；
- 算子的輸入、輸出、前置條件與效果；
- 算子如何組合；
- 組合何時閉合；
- 如何形成可驗證計算閉包；
- 同一算子如何具有文字、圖形與機器表示。

文字程式語言是計算史上極為成功的工具。

它具有：

- 低儲存成本；
- 高輸入效率；
- 精確語法；
- 成熟工具鏈；

- 容易搜尋、比較與傳輸。

  因此，後文本程式語言並不等於「沒有文字」。

  它真正意味的是：

> 程式的權威本體不再被限制為單一文字序列。

  程式可以先以語意結構存在，再投影為：

- 文字；
- 視覺節點；
- 狀態圖；
- 表格；
- UI；
- 契約；
- 測試；
- Runtime 計畫；
- AI 可查詢結構。

  這些投影不是彼此競爭的獨立真相，而是共同結構面向不同角色與任務的表達。

  本文將程式語言重新定義為：

> 【Programming Language
> =
> Constructible Structure
> +
> World Semantics
> +
> Executable Effects
> +
> Validation Rules】

而文字的位置是：

> 【Text
> =
> One Editable and Serializable Projection】

這個重新定位會改變程式設計方法。

開發者不再只是寫字，而是在：

- 建立節點；
- 連接世界身份；
- 限制合法組合；
- 宣告效果；
- 配置權限；
- 保留未決；
- 建立證據；
- 生成多重投影。

AI 也不應只作為更快的打字者。

它應協助：

- 從意圖展開結構；
- 找出歧義；
- 建立語意 IR；
- 生成候選投影；
- 驗證效果；
- 說明世界差分；

- 保留人類決策點。

因此，第 2 冊的起始命題是：

> 【結構先於文字，】

> 【語意先於執行，】

> 【驗證先於世界改變。】

當程式語言被理解為可構造世界結構，而非特殊文字，語言設計才可能真正進入意圖、AI、多投影與物理耦合的下一階段。

```
programming_language:
 language_id: "intent-structure-language"
 symbols:
  - "entity"
  - "state"
  - "action"
  - "event"
  - "effect"
  - "evidence"
 formation_rules:
  - "action requires actor and target"
  - "external effect requires authorization"
 structure:
  authority: "semantic-ir"
 semantics:
  world_binding: true
  identity_preserved: true
 effects:
  explicit: true
  irreversible_marked: true
 validation:
  - "type"
  - "state"
  - "permission"
  - "effect"
  - "evidence"
 runtime:
  multiple_backends: true
authoritative_structure:
```

```
  node_id: "payment-authorize-action"
  node_type: "action"
  semantics:
    actor: "order-module"
    target: "payment-attempt"
    precondition: "order.payment_status == pending"
  effects:
    external:
      - "request bank authorization"
    events:
      - "PaymentAuthorized"
      - "PaymentDeclined"
      - "PaymentOutcomeUnknown"
  projections:
    text:
      language: "TypeScript"
    visual:
      view: "payment-workflow"
    contract:
      format: "OpenAPI-plus-semantics"
    test:
      generated: true
semantic_diff:
  change_id: "payment-rule-change-04"
  text_diff:
    lines_changed: 7
  semantic_changes:
    rule:
      before: "amount <= order.total"
      after: "amount == authoritative_order_total"
    effects:
      added:
        - "PaymentMismatchRejected"
    affected:
      contracts:
        - "authorize-payment-v2"
      views:
        - "payment-state-view"
      actors:
        - "customer"
        - "support-agent"
  migration_required: true
```

- PU-2-01　程式語言不等於文字：結構、語意與執行的基本分離

- PU-2-02　符號作為算子：從靜態字元到可組合計算閉包
- PU-2-03　語法—語意—效果：程式語言的三層存在結構
- PU-2-04　意圖中介表示：從自然意圖到多重可執行投影
- PU-2-05　可編譯世界：程式執行作為世界狀態差分
- PU-2-06　後文本程式語言：意圖、結構、驗證與物理耦合的統一框架

SECTION

Neo.K／EveMissLab 相關理論

- Neo.K with Aletheia，《程式不等於程式碼：可執行問題模型的基本定義》，2026。
- Neo.K with Aletheia，《可視化作為外部結構記憶：多角色、多尺度的程式理解》，2026。
- Neo.K with Aletheia，《失敗也是程式：驗證、可觀測、恢復與長期維護》，2026。
- Neo.K with Aletheia，《結構先於文字：Nova 與後文本程式語言本體論》，2026。
- Neo.K with Aletheia，《符號作為算子：從靜態字元到可組合計算閉包》，2026。
- Neo.K with Aletheia，《可編譯世界：從程式執行到世界狀態演化》，2026。
- Neo.K with Aletheia，《意圖中介表示：從自然語言到多後端可驗證編譯》，2026。

SECTION

一般理論背景

- Backus, J., “Can Programming Be Liberated from the von Neumann Style?”, 1978.
- McCarthy, J., “Recursive Functions of Symbolic Expressions and Their Computation by Machine,” 1960.
- Scott, D. and Strachey, C., “Toward a Mathematical Semantics for Computer Languages,” 1971.
- Reynolds, J. C., Theories of Programming Languages, 1998.

- Pierce, B. C., Types and Programming Languages, 2002.
- Aho, A. V. et al., Compilers: Principles, Techniques, and Tools, 2006.
- Fowler, M., Domain-Specific Languages, 2010.
- Harel, D., “Statecharts: A Visual Formalism for Complex Systems,” 1987.
- Green, T. R. G. and Petre, M., “Usability Analysis of Visual Programming Environments,” 1996.
- Omar, C. et al., “Hazelnut: A Bidirectionally Typed Structure Editor Calculus,” 2017.
- Erdweg, S. et al., “The State of the Art in Language Workbenches,” 2013.
- Chomsky, N., Syntactic Structures, 1957.
- Knuth, D. E., “Literate Programming,” 1984.

SECTION

v0.1 — 2026-07-27

- 正式進入第 2 冊《程式語言的本質》。
- 完成十八篇地基論文第十三篇。
- 提出程式語言不等於文字。
- 建立符號、形成規則、結構、語意、效果、驗證與 Runtime 七元模型。
- 區分結構、語意與執行三層。
- 將文字定位為線性序列化投影。
- 區分字元、token、AST、語意結構、IR、執行計畫、效果與世界差分。
- 建立結構、語意與效果三重程式同一性。
- 分析文字與視覺語言的優勢及限制。
- 提出共同權威結構與多投影同步。

- 建立結構 Hole 與未完成程式合法狀態。
- 建立自然意圖、語意 IR、驗證與多重投影鏈。
- 將效果、權限、不可逆性與證據提升為語言一等結構。
- 擴展作用域至世界、資料、效果、時間與權限。
- 提出語意差分與擴展編譯模型。
- 加入條件、查詢、工作流、UI、AI Agent 與物理裝置案例。
- 提出二十二類失敗模式與十四項可證偽研究方向。
- 建立第 2 冊六篇完整路線。